

-1-

SPEECH-RELATED EVENT NOTIFICATION SYSTEM

The present application is based on and claims the benefit of U.S. provisional patent application Serial No. 60/219,861, filed July 20, 2000, the content of which is hereby incorporated by reference in its entirety.

INCORPORATION BY REFERENCE

The following Patent Application is hereby fully incorporated by reference, and priority is claimed therefrom; MIDDLEWARE LAYER BETWEEN SPEECH RELATED APPLICATIONS AND ENGINES, filed on December 29, 2000, Serial No. 09/751,836.

BACKGROUND OF THE INVENTION

The Present invention deals with services for enabling speech recognition and speech synthesis technology. In particular, the present invention relates to an event notification system in a middleware layer which lies between speech related applications and speech related engines.

Speech synthesis engines typically include a decoder which receives textual information and converts it to audio information which can be synthesized into speech on an audio device. Speech recognition engines typically include a decoder which receives audio information in the form of a speech signal and identifies a sequence of words from the speech signal.

In the past, applications which invoked these engines communicated directly with the engines.

Because the engines from each vendor interacted with applications directly, the behavior of that interaction was unpredictable and inconsistent. This made it virtually impossible to change synthesis or recognition engines without inducing errors in the application. It is believed that, because of these difficulties, speech recognition technology and speech synthesis technology have not quickly gained wide acceptance.

In an effort to make such technology more readily available, an interface between engines and applications was specified by a set of application programming interfaces (API's) referred to as the Microsoft Speech API version 4.0 (SAPI4). Though the set of API's in SAPI4 specified direct interaction between applications and engines, and although this was a significant step forward in making speech recognition and speech synthesis technology more widely available, some of these API's were cumbersome to use, required the application to be apartment threaded, and did not support all languages.

The process of making speech recognition and speech synthesis more widely available has encountered other obstacles as well. For example, the vendors of applications and engines have been required to write an enormous amount of code simply to implement the different interfaces for the different applications and engines that can be used together. In such systems, event notification is

very cumbersome. The engines are required to notify the applications directly of events, such as word boundaries, visemes, bookmarks, etc. This has required engines to know exactly how the application
5 wished to be notified of such events. Similarly, output devices (such as audio devices in a text-to-speech system) have also been required to know when events are occurring and how an application wishes to be notified of the events. Since applications
10 traditionally can be notified of events in one of a number of different ways, this required specific code to be written to interface to specific applications.

SUMMARY OF THE INVENTION

15 The present invention is directed to a system and method of notifying a speech related application of events generated by a speech related engine. A middleware layer receives a notification selection from the application. The notification
20 selection is indicative of a selected notification mechanism for notifying the application of the events. The middleware component receives an event indication from the engine. The event indication is indicative of an event generated by the engine. The
25 event indication is transferred to the application according to the selected notification mechanism.

In one embodiment, the event indication is first transferred to an output device, from the middleware component, which notifies the middleware

component when it has reached the event in an output data stream. The middleware component then retrieves the event indication from the output device and transmits it to the application.

5

In another embodiment, prior to transferring the event indication to the application, the middleware component receives an interest indication from the application indicative of events that the application is interested in. When the middleware component receives an event notification from the output device, it first determines whether the application is interested in the event by comparing the event to the interest indication received from the application.

In another embodiment, the middleware component initializes a notification assistance component based on the notification selection made by the application. The notification assistance component notifies the application of the event according to the selected notification mechanism.

The present invention can also be embodied as a method and apparatus for synchronizing a speech related output with processing within an application. A speech related data stream to be processed is received at a middleware component between the engine and the application. The speech related data stream is transferred from the application to the speech related engine and is processed by the engine. Event

indications from the engine are received at the middleware component. The event indications contain information that identifies an event and a position of the event in the data stream. The middleware
5 component transmits the event and position to the application.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a general
10 computing environment in which the present invention may be practiced.

FIG. 2 is a block diagram of a speech-related system in accordance with the present invention.

15 FIG. 3 is a more detailed block diagram of a TTS middleware component shown in FIG. 2.

FIG. 4 is a flow diagram illustrating the general operation of the system shown in FIG. 3.

FIG. 5 is a more detailed block diagram
20 illustrating the primary components used in an event notification system in accordance with one embodiment of the present invention.

FIG. 6 is a more detailed block diagram illustrating the primary components of the
25 notification system in accordance with another embodiment of the present invention.

FIG. 7 is a flow diagram illustrating how notification assistance object is initialized.

FIG. 8 is a flow diagram illustrating the general operation of the notification system shown in Figures 5 and 6.

FIG. 9 is a more detailed block diagram illustrating the primary components of a notification system of the present invention as applied to a speech recognition system, as opposed to a speech synthesis system.

DETAILED DESCRIPTION OF THE ILLUSTRATIVE EMBODIMENTS

FIG. 1 illustrates an example of a suitable computing system environment 100 on which the invention may be implemented. The computing system environment 100 is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the invention. Neither should the computing environment 100 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary operating environment 100.

The invention is operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well known computing systems, environments, and/or configurations that may be suitable for use with the invention include, but are not limited to, personal computers, server computers, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer

electronics, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

5 The invention may be described in the general context of computer-executable instructions, such as program modules, being executed by a computer. Generally, program modules include routines, programs, objects, components, data
10 structures, etc. that perform particular tasks or implement particular abstract data types. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a
15 communications network. In a distributed computing environment, program modules may be located in both local and remote computer storage media including memory storage devices.

With reference to FIG. 1, an exemplary
20 system for implementing the invention includes a general purpose computing device in the form of a computer 110. Components of computer 110 may include, but are not limited to, a processing unit 120, a system memory 130, and a system bus 121 that
25 couples various system components including the system memory to the processing unit 120. The system bus 121 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a

variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, 5 Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus also known as Mezzanine bus.

Computer 110 typically includes a variety of computer readable media. Computer readable media 10 can be any available media that can be accessed by computer 110 and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media 15 and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, 20 program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, 25 magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computer 100. Communication media typically embodies computer readable instructions,

data structures, program modules or other data in a modulated data signal such as a carrier WAV or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, FR, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media.

The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, FIG. 1 illustrates operating system 134, application programs 135, other program modules 136, and program data 137.

The computer 110 may also include other removable/non-removable volatile/nonvolatile computer

storage media. By way of example only, FIG. 1 illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156 such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through a non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150.

The drives and their associated computer storage media discussed above and illustrated in FIG. 1, provide storage of computer readable instructions, data structures, program modules and other data for the computer 110. In FIG. 1, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135,

other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a
5 minimum, they are different copies.

A user may enter commands and information into the computer 110 through input devices such as a keyboard 162, a microphone 163, and a pointing device 161, such as a mouse, trackball or touch pad. Other
10 input devices (not shown) may include a joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus, but
15 may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video
20 interface 190. In addition to the monitor, computers may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 190.

The computer 110 may operate in a networked
25 environment using logical connections to one or more remote computers, such as a remote computer 180. The remote computer 180 may be a personal computer, a hand-held device, a server, a router, a network PC, a peer device or other common network node, and

typically includes many or all of the elements described above relative to the computer 110. The logical connections depicted in FIG. 1 include a local area network (LAN) 171 and a wide area network (WAN) 173, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN networking environment, the computer 110 is connected to the LAN 171 through a network interface or adapter 170. When used in a WAN networking environment, the computer 110 typically includes a modem 172 or other means for establishing communications over the WAN 173, such as the Internet. The modem 172, which may be internal or external, may be connected to the system bus 121 via the user input interface 160, or other appropriate mechanism. In a networked environment, program modules depicted relative to the computer 110, or portions thereof, may be stored in the remote memory storage device. By way of example, and not limitation, FIG. 1 illustrates remote application programs 185 as residing on remote computer 180. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

FIG. 2 is a more detailed block diagram of a speech system 200 in accordance with one embodiment

of the present invention. It should be noted that speech system 200 can be incorporated into the environment illustrated in FIG. 1. Speech system 200 includes one or more speech applications 202 (such as
5 speech recognition or speech synthesis applications), speech middleware component 204, one or more speech recognition engines 206 and/or one or more text-to-speech engines (synthesizers) 208.

In one illustrative embodiment, speech
10 middleware component 204 is implemented in the operating system 134 illustrated in FIG. 1. Speech middleware component 204, as shown in FIG. 2, includes speech recognition middleware component 210, context free grammar (CFG) engine 212 and text-to-
15 speech middleware component 214.

Briefly, in operation, speech middleware component 204 resides between applications 202 and engines 206 and 208. Applications 202 can be speech recognition and speech synthesis applications which
20 desire to invoke engines 206 and 208. In doing so, applications 202 make calls to speech middleware component 204 which, in turn, makes calls to the appropriate engines 206 and 208 in order to have speech recognized or synthesized.

25 For example, applications 202 may provide the source of audio data for speech recognition. Speech middleware component 204 passes that information to speech recognition engine 206 which simply recognizes the speech and returns a

recognition result to speech recognition middleware component 210. Speech recognition middleware component 210 places the result in a desired format and returns it to the application 202 which requested it or to another desired location specified by the application 202.

Similarly, an application 202 can provide a source of textual data to be synthesized. TTS middleware component 214 assembles that data, and provides it to TTS engine 208, for synthesis. TTS engine 208 simply synthesizes the data and returns audio information, along with associated event information, to TTS middleware component 214, which handles spooling of that information to an audio device, writing that information to memory, or placing that information in any other desired location, as specified by the application 202 which requested it.

CFG engine 212, briefly, assembles and maintains grammars that are to be used by speech recognition engine 206. This allows multiple applications and multiple grammars to be used with a single speech recognition engine 206.

FIG. 3 is a more detailed block diagram of a portion of system 200 shown in FIG. 2. Specifically, FIG. 3 illustrates TTS middleware component 214 in greater detail. TTS middleware component 214 illustratively includes a set of COM objects illustrated as the SpVoice object 216, Site

object 218 and lexicon container object 220. In addition, TTS middleware component 214 can optionally include a format converter object and an audio output object 224. In one illustrative embodiment, communication between the objects in TTS middleware component 214 and applications 202 is accomplished using application programming interfaces (APIs). Similarly, communication between the objects in TTS middleware component 214 and the TTS engine object 208 is accomplished using device driver interfaces (DDIs). One illustrative embodiment of DDIs and APIs and their related structures is set out in related patent application Serial No. 09/751,836 entitled MIDDLEWARE LAYER BETWEEN SPEECH RELATED APPLICATIONS AND ENGINES filed on December 29, 2000.

A general discussion of the operation of TTS middleware component 214, with applications 202 and engine 208, is illustrated by the flow diagram in FIG. 4. Initially, application 202 opens an instance of the SpVoice object 216. In one illustrative embodiment, the application calls the COM CoCreateInstance for the component CLSID_SpVoice to get a pointer to the interface ISpVoice of the SpVoice object. SpVoice object 216 then creates lexicon container object 220 and an XML parser object 228. This is indicated by blocks 230, 232 and 234 in FIG. 4.

Next, application 202 can specify the attributes of TTS engine 208, such as whether the

engine which is the synthesizer exhibits male or female voice qualities, the language of the synthesis, etc. This is done, for example, by calling the SetVoice method on the SpVoice object 216. This is indicated by optional block 236 in FIG. 4. In addition, the application can optionally specify the particular audio output object 224 which is desired. This is indicated by optional block 238 in FIG. 4.

The application 202 can set other attributes associated with the voice speaking, such as the rate and volume of speech, using for example, the SetRate and the SetVolume methods exposed by the SpVoice object 216. These are optional as well.

It should be noted that specifying the attributes of the engine 208 and audio output object 224 are optional. If the application does not specify these items, the first call to the SpVoice object 216 requiring synthesis results in the SpVoice object 216 choosing and initializing the default voice (i.e., the default TTS engine 208) and the default audio output object 224.

Application 202 must then indicated to SpVoice object 216 which particular events it is interested in, and how it wishes to be notified of those events. This will be discussed in greater detail below. Suffice it to say that application 202 will likely be interested in a variety of different events and will wish to be notified when those events

are occurring (or prior to the occurrence) at audio output object 224. Similarly, application 202 may wish to be notified of the events in one of a variety of different manners. Therefore application 202
5 provides this information to SpVoice object 216. This is indicated by block 239.

Once these items are configured properly, application 202 can call the SpVoice object 216 and request that textual information be synthesized.
10 This can be done, for example, by calling the Speak or the SpeakStream methods on the SpVoice object 216. This is indicated by block 240.

The SpVoice object 216 then performs format negotiation. This does not form part of the present
15 invention and is only optional. Briefly, the SpVoice object 216 attempts to optimize the format of data created by TTS engine 208 and that accepted by audio output object 224 for optimal synthesis. Format negotiation is indicated by block 242 in FIG. 4.

The SpVoice object 216 then breaks the textual information provided by application 202 into text fragments. For example, if the textual information is in XML, the SpVoice object 216 invokes the XML parser 228 to parse the XML input into text
20 fragments. While the textual information can come from a variety of sources (such as a text buffer, straight textual information, XML, etc.) that information is broken into text fragments by SpVoice object 216, as indicated by block 244 in FIG. 4.
25

The SpVoice object 216 then calls a speak method on TTS engine 208, passing in the information to be synthesized. This is indicated by block 246. In doing this, the SpVoice object 216 also specifies
5 a Site object 218 to be used by the TTS engine for returning the synthesized information.

TTS engine 208 receives the text fragments, synthesizes the text into PCM (pulse code modulation) data (or other suitable audio data) and provides an
10 indication of where events occur in the PCM data. For example, TTS engine 208 can illustratively provide an indication of where word and phoneme boundaries occur in the PCM data. This information is all provided from TTS engine 208 to SpVoice object
15 216 through the Site object 218.

It should be noted that, in performing the synthesis, TTS engine 208 can access the lexicon object 220 contained in TTS middleware component 214. The lexicon container object also forms no part of
20 the present invention and is mentioned only for the sake of completeness. Briefly, the lexicon container object 220 contains all lexicons of interest and the TTS engine 208 simply needs to access object 220 as if it were a single lexicon.

25 Synthesizing the actual fragments and writing them and the events to the Site object are indicated by blocks 248 and 250 in FIG. 4.

During the format negotiation step 242, the SpVoice object 216 determines whether the format of

the audio output object 224 or the format of the information provided by TTS engine 208 need to be converted. If conversion is required, information is provided to a format converter object, such as
5 through the ISpAudio or ISpStream interfaces, where the information is converted into a desired format for the audio output object 224. The format converter object then manages the process of spooling out the audio information to audio output object 224
10 and also manages returning events noticed by the audio output object 224 to the Site object 218 and the SpVoice object 216 for transmission back to the application 202. Where no format conversion is desired, the information from the Site object 218 is
15 spooled out to the audio output object 224 by the SpVoice object 216, through a suitable interface such as the ISpStream interface, and the audio output object 224 returns events to the SpVoice object. This is indicated by blocks 256 and 258.

20 Of course, it should also be noted that rather than providing the information directly to an audio output object 224, the information can be written to memory, or provided at some other specified output or location.

25 In notifying the SpVoice object 216 of events, audio output object 224 can do one of any number of things. For example, audio output object 224 can provide a notification that an event has occurred, or is occurring, as audio output object 224

is playing the data containing the event. However, it may well be that application 202 wishes to be notified of the event prior to the occurrence of the event at the audio object. For example, if the event
5 is a viseme, application 202 may desire to be notified of the viseme so that it can animate a character on a computer screen, as the audio information is being played. In that instance, or in any instance where application 202 desires to be
10 notified of the event prior to it occurring, audio output object 224 can be configured to notify the SpVoice object 216 that an event is about to occur at any predetermined offset prior to the event in the data stream. This simply requires audio output
15 object 224 to look forward in the data stream by the offset amount and notify the SpVoice object 216 when an event is encountered.

As will be described later with respect to Figures 6, 7 and 9, the SpVoice object 216 (or its
20 counterpart in SR middleware component 210) can initialize a notification assistance component whose primary job is to notify the application 202 of events. Therefore, when the SpVoice object 216 receives the event from audio output object 224, it
25 first determines whether application 202 is even interested in the event. This was indicated at block 239 when the application indicated to SpVoice object 216 the particular types of events it was interested in.

Assuming that application 202 is interested in the event which has been notified, then SpVoice object 216 may notify application 202 directly of the event, using the specific notification mechanism selected by the application in step 239 of FIG. 4. However, when the notification assistance object has been initialized, the SpVoice object 216 notifies the notification assistance object that an event which the application is interested in has occurred. The notification assistance object then notifies the application 202 of the event. This is indicated by block 260.

FIG. 5 is a more detailed block diagram of components in the system shown in FIG. 3, which are used in notifying application 202 of events. A number of the items shown in FIG. 5 are the same as those shown in FIG. 3 and are similarly numbered. However, FIG. 5 also illustrates that audio object 204 includes an event queue 300 and a completed queue 302. As described in greater detail below, when events are provided to audio output object 204, they are placed in event queue 300. When they are reached by audio output object 204, they are placed in the completed queue 302, and then the audio output object 204 notifies SpVoice 216 of those events. FIG. 5 shows that the various objects in the system support a number of interfaces. Each interface exposes one or more methods, in order to implement the notification system. The data (such as data to be

synthesized, and synthesized audio data) is simply shown by arrows. However, it will of course be appreciated that the data is illustratively passed among objects by invoking methods on interfaces exposed by the objects. Those interfaces are not discussed in great detail here since they do not form part of the present invention.

FIG. 6 is similar to FIG. 5, and similar items are numbered the same. However, FIG. 6 shows an embodiment in which a notification assistant object 304 has been initialized as well.

FIG. 7 is a flow diagram illustrating the initialization of notification assistance object 304. The application 202 calls SpVoice object 216 to select a notification mechanism for notifying the application of events. For example, application 202 can call the SetNotifyxxxx method on the ISpNotifySource interface 307 of SpVoice object 216. The xxxx designates the particular notification mechanism by which application 202 wishes to be notified.

For example, if the application 202 calls SetNotifySink that indicates that the application is implementing an ISpNotifySink which exposes a Notify method that can simply be called by notification assistance object 304 to notify application 202 of an event.

If the application invokes SetNotifyCallbackFunction that indicates that

application 202 simply wishes to be notified using a callback function, such as those common in the C or C++ programming languages.

If application 202 invokes
5 SetNotifyWindowMessage that indicates that
application 202 wishes to be notified using a window
callback function to receive notifications by having
a window message posted.

If the application calls
10 SetNotifyWin32Event this simply sets up a Win32 event
object and indicates that the application supports
Win32 event notification and can be notified
according to that mechanism.

These are but a few of the possible
15 notification mechanisms that can be selected. The
application 202 notifying SpVoice 216 of its selected
notification mechanism is indicated by block 306.

Application 202 then provides an indication
as to the particular types of events it is interested
20 in to SpVoice Object 216. This is indicated by block
308. In doing so, the application may illustratively
invoke the SetInterests method on the ISpEventSource
interface 310 on SpVoice object 216. This method
allows the application 202 to specify the different
25 event types which it wishes to be notified of.

Once SpVoice object 216 has received the
selected notification mechanism from application 202,
it initializes notification assistance object 304
according to the notification mechanism selected by

the application. This is indicated by block 312. This can be done, for example, by initializing assistance object 304 by invoking the Initxxxx method on the ISpNotifyTranslator interface. Notification
5 assistance object 304 is thus initialized such that, when SpVoice object 216 invokes the Notify method on the ISpNotifySink interface 314 supported by notification assistance object 304 notifies application 202 according to the selected method. In
10 one illustrative embodiment, notification assistance object 304 supports an interface 318 which makes the events available for retrieval and examination by application 202. In another embodiment, notification assistance object 304 simply notifies application 302
15 of the event and application 202 retrieves or examines the event from the SpVoice object 216.

FIG. 8 is a flow diagram which illustrates the operation of the event notification systems shown in FIGS. 5 and 6. FIG. 8 presumes that TTS engine
20 208 has received data to be synthesized and is synthesizing that data. TTS engine 208 thus identifies events and an offset into the data stream where the event is located. This is indicated by block 350.

25 TTS engine 208 invokes the AddEvents method on interface 352 of the Site object 218. This adds the event identified at block 350 by the TTS engine to the Site object 218. This is indicated by block 354 in FIG. 8.

The SpVoice object 216 then transfers the event and its offset in the audio stream to audio object 204. The event and its offset are placed in event queue 300 in audio object 204. Of course,
5 transferring of this event can be accomplished through appropriate interfaces as well. Transferring the event and its offset to event queue 300 in audio object 204 is indicated by block 356 in FIG. 8.

Audio object 204 begins playing the data.
10 As it plays the data, it looks a predetermined offset distance ahead in the data stream to determine whether any events are upcoming in event queue 300. When the audio object encounters an event in the data stream, it transfers the event and its offset from
15 the event queue 300 to its completed queue 302. After this is accomplished it invokes the Notify method on the ISpNotifySink interface 358 on the SpVoice object 216. This is indicated by block 360 in FIG. 8. This notifies SpVoice object 216 that
20 audio object 204 has reached an event in its event queue.

After being notified of an event, SpVoice object 216 invokes the GetEvents method on the ISpEventSource interface 362 of audio object 204.
25 This is indicated by block 364 in FIG. 8. Audio object 204 then returns to the SpVoice object 216 the contents of its completed events queue 302. This is indicated by block 366.

Upon receiving the event information indicative of the particular event about which SpVoice object 216 was notified, SpVoice object 216 determines whether an application 202 is even
5 interested in being notified of this event. Recall that application 202 indicated the events in which it was interested to SpVoice object 216 by invoking the SetIntersts method on the ISpEventSource interface 310. Therefore, SpVoice object 216 simply needs to
10 compare the event type which was returned from audio object 204 with those specified as being of interest to application 202. This is indicated by blocks 368 and 370 in FIG. 8. Of course, if the application is not interested in this event, the SpVoice object
15 updates it's internal state and then the event is simply discarded and SpVoice object 216 awaits notification of the next event encountered by audio output object 204. This is indicated by blocks 372 and 374.

20 If SpVoice object 216 determines that application 202 is interested in this event, then what happens next depends on whether the notification system is an embodiment in which the notification assistance object 304 is implemented. If not, the
25 processing simply proceeds at block 376 and SpVoice object 216 simply notifies application 202 according to the notification mechanism which it selected at the outset.

However, if notification assistance object 304 is implemented (as shown in FIG. 6) then SpVoice object 216 indicates to object 304 that an event has been encountered. In one illustrative embodiment, 5 SpVoice object 216 does this by invoking the Notify method on the ISpNotifySink interface 314 supported by notification assistance object 304. This is indicated by block 378 in FIG. 8. At that point, notification assistance object 304 notifies 10 application 202 according to the selected notification method, for which it was initialized. This is indicated by block 376. In one embodiment, notification assistance object retrieves the event from the SpVoice object and makes it available to the 15 application, as indicated by block 380. In another embodiment, the notification assistance object simply notifies the application of the event and the application accesses the SpVoice object for the event.

20 FIG. 9 is a more detailed block diagram illustrating a notification system in conjunction with speech recognition engine 206. A number of items are similar to those shown in FIGS. 5 and 6 and are similarly numbered. Of course, the event 25 notification system can be implemented with or without notification assistance object 304. It is shown with notification assistance object 304 for the sake of completeness.

SpRecoContext object 400 is the counterpart to the SpVoice object 216 in TTS middleware component 214 in that it generally manages data flow and performs services within SR middleware component 210.

5 The SpRecoContext object 400 exposes one or more interfaces that can be used with application 202. SpRecoContext object 400 also calls interface methods exposed by SR engine object 206. A more detailed discussion of the SpRecoContext object 400 can be

10 found in the above-identified and related patent application. However, no further explanation is needed for the sake of the present invention.

Processing events is highly similar to that with respect to the TTS systems shown in FIGS. 5 and

15 6. In other words, SR engine 206 receives voice data to be recognized. The recognition results are provided to Site object 218. Also, events (such as recognitions, word boundaries, phoneme boundaries, etc.) are added to Site object 218 by invoking the

20 AddEvents methods. These items are provided back to the SpRecoContext object 400. In response, in the embodiment in which notification assistance object 304 is not present, SpRecoContext object 400 simply determines whether application 202 is interested in

25 the events (because the application has previously notified SpRecoContext object 400 of the events in which it is interested) and, if so, notifies application 202 by calling the applicatoion's Notify method.

In the embodiment shown in FIG. 9, in which the notification assistance object 304 is implemented, SpRecoContext object 400 calls Notify on the ISpNotifySink interface 314 supported by object 304. This indicates to object 304 that SpRecoContext object 400 has encountered an event. In response, the notification assistance object 304 notifies application 202 according to its selected notification mechanism, by which notification assistance object 304 was initialized. Of course, where an input object is used with SpRecoContext object 400, interaction is similar to that with respect to the audio output object 204 in SpVoice object 216.

Appendix A illustrates a number of the interfaces discussed herein in greater detail, simply for the sake of completeness. They can be implemented in other ways as well and still comport with the inventive features of the present invention.

It can thus be seen that the present invention is directed to a middleware layer that is arranged between the applications and engines. The middleware layer supports interfaces that allow the applications to select one of a variety of different types of notification mechanisms. The middleware layer then configures itself to provide event notification from the engine to the application in the selected manner. In addition, the present invention allows an output device to simply notify

the middleware layer when it has encountered the location in the output data stream which coincides with an event. The middleware layer then takes care of notifying the application, and the output device
5 need not even be aware of the type of the event for which notification was sent. This also significantly simplifies the coding required to make output devices consistent with the other components in the system.

Although the present invention has been
10 described with reference to preferred embodiments, workers skilled in the art will recognize that changes may be made in form and detail without departing from the spirit and scope of the invention.

APPENDIX A

15

ISpNotifySource

In both speech synthesis and speech recognition, applications receive notifications when words have been spoken or when phrases have been recognized.
20 SAPI components that generate notifications implement an ISpNotifySource.
The ISpNotifySource and ISpNotifySink interfaces by themselves only provide a mechanism for a notification but no information on the events that
25 caused the notification. With an ISpEventSource object, an application can retrieve information about the events that caused the notification. An ISpEventSource also provides the mechanism to filter and queue events. By default, an application (really
30 an ISpNotifySink) receives no notifications from ISpEventSource until SetInterests has been called to specify on which events to notify or queue.
When an application is notified of an event that is not queued, an application will take measures based
35 on which event sink receives the notification. From

context, an application might know exactly what it needs to do, or it may need to interact with the components which sent the notifications. If an application is notified of an event which is queued, then the application will call `ISpEventSource::GetEvents` to retrieve the actual events that caused a notification.

When to Implement

Implement the `ISpNotifySource` interface during initialization to set the default action for how an event source notifies the receiver.

Methods in Vtable Order

ISpNotifySource Methods	Description
<u>SetNotifySink</u>	Sets up the instance to make free-threaded calls through <code>ISpNotifySink::Notify</code> .
<u>SetNotifyWindowMessage</u>	Sets a window callback function to receive notifications as window messages.
<u>SetNotifyCallbackFunction</u>	Sets a callback function to receive notifications.
<u>SetNotifyCallbackInterface</u>	Enables an object derived from <code>ISpTask</code> to receive notifications.
<u>SetNotifyWin32Event</u>	Sets up a Win32 event object to be used by this instance.
<u>WaitForNotifyEvent</u>	A blocking call in response to a SAPI notification event.
<u>GetNotifyEventHandle</u>	Retrieves notifications via Win32 events.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifySource::SetNotifySink

`ISpNotifySource::SetNotifySink` sets up the instance to make free-threaded calls through `ISpNotifySink::Notify`.

```
HRESULT SetNotifySink(  
    ISpNotifySink *pNotifySink  
);
```

Parameters

pNotifySink

[in] Pointer to the notification method. May be NULL if no default action is required.

Return values

Value	Description
S_OK	Function completed successfully.
E_FAIL	Interface not initialized.
FAILED (hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

5

ISpNotifySource::SetNotifyWindowMessage

ISpNotifySource::SetNotifyWindowMessage sets up the instance to send window messages to a specified window.

10

HRESULT SetNotifyWindowMessage(

HWND *hWnd*,
UINT *Msg*,
WPARAM *wParam*,
LPARAM *lParam*

15

);

Parameters

hWnd

20

[in] Handle to the window whose message handler function will receive SAPI notifications.

Msg

[in] Message number which will be passed into the message handler function of the window *hWnd*.

wParam

25

[in] *wParam* that will be passed into the message handler function of the window *hWnd*.

lParam

[in] *lParam* that will be passed into the message handler function of the window *hWnd*.

1.0.0.0 "E:\SAP\T300"

Return values

	Value	Description
	S_OK	Function completed successfully.
	E_FAIL	Interface not initialized.
	FAILED (hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifySource::SetNotifyCallback Function

- 5 **ISpNotifySource::SetNotifyCallbackFunction** sets up this instance to send notifications via a standard C-style callback function.

```
10 HRESULT SetNotifyCallbackFunction(  
    SPNOTIFYCALLBACK *pfnCallback,  
    WPARAM            wParam,  
    LPARAM            lParam  
);
```

Parameters

pfnCallback

- 15 [in] The notification callback function to be used.

wParam

[in] Constant word value that will be passed to the *pfnCallback* function when it is called.

lParam

- 20 [in] Constant long value that will be passed to the *pfnCallback* function when it is called.

Return values

	Value	Description
	S_OK	Function completed successfully.
	E_FAIL	Interface not initialized.
	FAILED (hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifySource::SetNotifyCallbackInterface

ISpNotifySource::SetNotifyCallbackInterface sets up this instance to call the virtual method **ISpNotifyCallback::NotifyCallback** for notifications.

```
5 HRESULT SetNotifyCallbackInterface(  
    ISpNotifyCallback *pSpCallback,  
    WPARAM wParam,  
    LPARAM lParam  
10 );
```

Parameters

pSpCallback

[in] A pointer to an application-defined implementation of the ISpNotifyCallback interface.

15 *wParam*

[in] Constant word value that will be passed to the NotifyCallback method when it is called.

lParam

20 [in] Constant long value that will be passed to the NotifyCallback method when it is called.

Return values

Value	Description
S_OK	Function completed successfully.
E_FAIL	Interface not initialized.
FAILED (hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifySource::SetNotifyWin32Event



25 **ISpNotifySource::SetNotifyWin32Event** sets up a Win32 event object to be used by this instance.

For an explanation of Win32 event objects, see the Win32 Platform SDK documentation. Once an event
30 object has been initialized for this instance, use

either the WaitForNotifyEvent and
GetNotifyEventHandle methods. Note that Win32 event
objects and SAPI events are different notifications.
HRESULT SetNotifyWin32Event (void);

5 **Parameters**

None

Return values

Value	Description
S_OK	Function completed successfully.
E_FAIL	Interface not initialized.
FAILED (hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifySource::WaitForNotifyEvent

10 **nt**

ISpNotifySource::WaitForNotifyEvent is a blocking
call in response to a SAPI notification event.
A blocking call returns when a SAPI notification has
fired, a timeout has passed or the initialized Win32
event object has signaled. This call is only valid
after calling InitWin32Event.

HRESULT WaitForNotifyEvent(
 DWORD dwMilliseconds
);

20 **Parameters**

dwMilliseconds

[in] Number of milliseconds for the timeout on a blocking call. If set to
INFINITE, there is no timeout.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	InitWin32Event did not return successfully or has not been called.
E_FAIL	Interface not initialized.
FAILED (hr)	Appropriate error message.

25 © 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifySource::GetNotifyEventHandle

ISpNotifySource::GetNotifyEventHandle retrieves the Win32 event object handle.

5 **HANDLE** GetNotifyEventHandle (void);

Parameters

None

Return values

Value	Description
Win32 event object	Initialized by InitWin32Event on this <u>ISpNotifyTranslator</u> instance.
NULL	Interface not initialized.
FAILED (hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

10

ISpNotifySink



In both speech synthesis and speech recognition, applications receive notifications when words have been spoken or when phrases have been recognized.

15 SAPI components that generate notifications implement an ISpNotifySource.

The ISpNotifySource and ISpNotifySink interfaces by themselves only provide a mechanism for a notification but no information on the events that caused the notification. With an ISpEventSource object, an application can retrieve information about the events that caused the notification. An

20 ISpEventSource also provides the mechanism to filter and queue events. By default, an application (really an ISpNotifySink) receives no notifications from ISpEventSource until SetInterests has been called to specify on which events to notify or queue.

25 When an application is notified of an event which is not queued, an application will take measures based on which event sink is receiving the notification.

30 From context an application might know exactly what it needs to do, or it may need to interact with the components which sent the notifications. If an

application is notified of an event which is queued, then the application will call `ISpEventSource::GetEvents` to retrieve the actual events that caused a notification.

5 **When to Implement**

Implement the `ISpNotifySink` interface when an `ISpNotifySink` object is to be notified.

Methods in Vtable Order

ISpNotifySink Methods	Description
<u>Notify</u>	Notifies the <code>ISpNotifySink</code> object.
© 1995-2000 Microsoft Corporation. All rights reserved.	

10

ISpNotifySink::Notify

ISpNotifySink::Notify notifies an `ISpNotifySink` object when an event has occurred.

15 If a message has not already been posted, this method either sets an event or posts a message to the private window. Often an application will call specific status functions based on the context of where a notification has come from. For instance, an application receiving a notification from an `ISpVoice` 20 instance can call `ISpVoice::GetStatus` to find out the most recent cause of a `Notify` call.

HRESULT Notify (void);

Parameters

None

25 **Return values**

Value	Description
<code>S_OK</code>	Function completed successfully.
<code>SPERR_UNINITIALIZED</code>	Object has not been properly initialized.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyTranslator

ISpNotifyTranslator inherits from ISpNotifySink. The component CLSID_SpNotify, provides this interface for reuse by implementers of the ISpNotifySource interface. It provides a proxy object to other calls so that a developer does not need to re-address threading issues. Many, but not all, of these methods are identical to those in ISpNotifySource.

When to Use

ISpNotifyTranslator may be used in applications to pass in specific Win32 events.

Methods in Vtable Order

ISpNotifyTranslator Methods	Description
<u>InitWindowMessage</u>	Enables a window callback function to receive notifications as window messages.
<u>InitCallback</u>	Enables a callback function to receive notifications.
<u>InitSpNotifyCallback</u>	Enables an object derived from ISpTask to receive notifications.
<u>InitWin32Event</u>	Sets up a Win32 event object to be used by this instance.
<u>Wait</u>	A blocking call in response to a SAPI notification event.
<u>GetEventHandle</u>	Retrieves notifications via Win32 events.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyTranslator::InitWindowMessage

ISpNotifyTranslator::InitWindowMessage sets up the instance to send window messages to a specified window.

```
HRESULT InitWindowMessage(  
    HWND    hWnd,  
    UINT    Msg,  
    WPARAM  wParam,
```

```
    LPARAM    lParam
);
```

Parameters

hWnd

- 5 [in] Handle to the window whose message handler function will receive SAPI notifications.

Msg

[in] Message number which will be passed into the message handler function of the window *hWnd*.

10 *wParam*

[in] *wParam* that will be passed into the message handler function of the window *hWnd*.

lParam

- 15 [in] *lParam* that will be passed into the message handler function of the window *hWnd*

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Interface is already initialized.
E_INVALIDARG	<i>hWnd</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyTranslator::InitCallback

- 20 **ISpNotifyTranslator::InitCallback** sets up this instance to send notifications via a standard C-style callback function.

```
HRESULT InitCallback(
    SPNOTIFYCALLBACK *pfnCallback,
    WPARAM            wParam,
    LPARAM            lParam
);
```

Parameters

pfnCallback

[in] The notification callback function to be used.

wParam

[in] Constant word value that will be passed to the *pfnCallback* function when it is called.

lParam

- 5 [in] Constant long value that will be passed to the *pfnCallback* function when it is called.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Interface is already initialized.
E_INVALIDARG	<i>pfnCallback</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyTranslator::InitSpNotifyCa llback

10

ISpNotifyTranslator::InitSpNotifyCallback sets up this instance to call the virtual method ISpNotifyCallback::NotifyCallback for notifications.

- 15 **HRESULT InitSpNotifyCallback**(
 ISpNotifyCallback *pSpCallback,
 WPARAM wParam,
 LPARAM lParam
);

Parameters

- 20 *pSpCallback*

[in] A pointer to an application-defined implementation of the ISpNotifyCallback interface.

wParam

- 25 [in] Constant word value that will be passed to the NotifyCallback method when it is called.

lParam

[in] Constant long value that will be passed to the NotifyCallback method when it is called.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Interface is already initialized.
E_INVALIDARG	<i>pSpNotifyCallback</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyTranslator::InitWin32Event

5 **ISpNotifyTranslator::InitWin32Event** sets up a Win32 event object to be used by this instance. This method is applicable only with objects using Win32 events. For an explanation of Win32 event objects see the Win32 Platform SDK documentation.

10 Once an event object has been initialized for this instance, then use `WaitForNotifyEvent` and `GetNotifyEventHandle` methods. Win32 event objects and SAPI events are different. It is identical to `ISpNotifySource::SetNotifyWin32Event` except with two

15 additional parameters.

```
HRESULT InitWin32Event(  
    [in] HANDLE    hEvent,  
    [in] BOOL      fCloseHandleOnRelease  
);
```

20 Parameters

hEvent

Handle of an existing Win32 event object for the application to use with `ISpNotifyTranslator`. An `ISpNotifyTranslator` object will take care of all Win32 event object details. May be NULL, in which case an application

25 may call `ISpNotifyTranslator::Wait` to block a thread until a SAPI notification occurs.

fCloseHandleOnRelease

Specifies whether the *hEvent* handle should be closed when the object is released. If *hEvent* is NULL, then this ignore this parameter and always

30 close the handle upon release of the object.

Return values

Value	Description
-------	-------------

S_OK Function completed successfully.
SPERR_ALREADY_INITIALIZED Interface is already initialized.
FAILED(hr) Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyTranslator::Wait

ISpNotifyTranslator::Wait is a blocking call in response to a SAPI notification event.

- 5 A blocking call returns when a SAPI notification has fired, a timeout has passed or the initialized WIN32 event object has signaled. This method is applicable only with objects using Win32 events.

10 **HRESULT Wait**(
 DWORD *dwMilliseconds*
);

Parameters

dwMilliseconds

- 15 [in] Number of milliseconds for the timeout on a blocking call. If set to INFINITE, there is no timeout.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The event was not set and the call was timed out.
SPERR_UNINITIALIZED	InitWin32Event did not return successfully or has not been called.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyTranslator::GetEventHandle

- 20 **ISpNotifyTranslator::GetEventHandle** returns the Win32 event object handle initialized by InitWin32Event on this ISpNotifyTranslator instance. This method is applicable only with objects using Win32 events. The handle is not a duplicated handle and should not be closed by the caller.

25 **HANDLE GetEventHandle** (void);

Parameters

None

Return values

	Value	Description
	<i>handle</i>	The handle to the event
	INVALID_HANDLE_VALUE	Call failed.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpEventSink

- 5 This interface allows event sources to send events directly to an event sink through a free-threaded call.

When to Implement

This interface is never used by most applications.

Methods in Vtable Order

	ISpEventSink Methods	Description
	<u>AddEvents</u>	Adds events directly to an event sink.
	<u>GetEventInterest</u>	Passes back the event interest for the voice.

10 © 1995-2000 Microsoft Corporation. All rights reserved.

ISpEventSink::AddEvents

ISpEventSink::AddEvents adds events directly to an event sink.

15 HRESULT AddEvents(
 const SPEVENT *pEventArray,
 ULONG ulCount
);

Parameters

pEventArray

- 20 Pointer to an array of SPEVENT event structures.

ulCount

 Number of event structures being passed in.

Return values

	Value	Description
	S_OK	Function completed successfully.
	E_INVALIDARG	<i>pEventArray</i> is bad or invalid
	FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpEventSink::GetEventInterest

ISpEventSink::GetEventInterest passes back the event interest for the voice.

5 HRESULT GetEventInterest(
 ULONGLONG *pullEventInterest
);

Parameters

pullEventInterest

10 [out] Set of flags of type SPEVENTENUM defining the event interest.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Pointer bad or invalid.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpEventSource



15 This interface provides functionality for events which can be queued, filtered or can cause a notification to ISpNotifySink.
The ISpEventSource inherits from the ISpNotifySource interface.

Methods in Vtable Order

ISpEventSource Methods	Description
<u>SetInterest</u>	Sets the types of events.
<u>GetEvents</u>	Retrieves and removes the queued events.
<u>GetInfo</u>	Returns queuing and interest information about the event.

20 © 1995-2000 Microsoft Corporation. All rights reserved.

ISpEventSource::SetInterest

ISpEventSource::SetInterest sets the type of events which will invoke a notification and become queued.

25 If SetInterest is never called, the speech recognition engine defaults to SPEI_RECOGNITION as

the sole event interest. No events will be passed through if both parameters are set to zero.

```
HRESULT SetInterest(  
    ULONGLONG    ullEventInterest,  
5    ULONGLONG    ullQueuedInterest  
);
```

Parameters

ullEventInterest

10 [in] Event ID flags indicating which events should invoke a notification to the event sink that this event source uses.

ullQueuedInterest

[in] Event ID flags indicating which events should be queued prior to ISpEventSource::GetEvents. The event flags set here must also be set in *dwEventInterest*.

15 Return values

Value	Description
S_OK	Function completed successfully.
E_FAIL	Interface not defined.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpEventSource::GetEvents

ISpEventSource::GetEvents retrieves and removes the events which have been queued.

```
20 HRESULT GetEvents(  
    ULONG    ulCount,  
    SPEVENT *pEventArray,  
    ULONG    *pulFetched  
);
```

25 Parameters

ulCount

[in] Maximum number of events that SPEVENT structures can return.

pEventArray

30 [out] Pointer to array of SPEVENT structures. Each returned event is written to one of these SPEVENT structures.

pulFetched

- 5 [out] Pointer to the number of events returned. This number represents the earliest events to take place. These events are then removed from the queue. The events not returned are left for a future call to GetEvents. It is possible that by the time an application calls GetEvents, another thread has processed the events and there are no events to be returned. This may be the result of subsequent Notify calls.

Return values

Value	Description
S_OK	Function completed successfully.
E_FAIL	Interface not valid.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpEventSource::GetInfo

- 10 **ISpEventSource::GetInfo** passes back the information about the event.

```
HRESULT GetInfo(  
    SPEVENTSOURCEINFO *pInfo  
);
```

- 15 **Parameters**

pInfo

[out] Pointer to a SPEVENTSOURCEINFO structure about the event.

Return values

Value	Description
S_OK	Function completed successfully.
E_FAIL	Interface not valid.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpNotifyCallback

Note: This is not a COM interface.

Methods in Vtable Order

ISpNotifySource Methods	Description
<u>NotifyCallback</u>	Sets the notification mechanism for a particular instance.

© 1995-2000 Microsoft Corporation. All rights reserved.

<http://www.microsoft.com/iit/>



ISpNotifyCallback::NotifyCallback

ISpNotifyCallback::NotifyCallback sets the notification mechanism for a particular instance.

This method is not required to be defined and implementation is unique to the application.

HRESULT NotifyCallback (

WPARAM *wParam*,

LPARAM *lParam*

);

Parameters

wParam

[in] *wParam* that will be passed into the message handler function of the window *hWnd*.

lParam

[in] *lParam* that will be passed into the message handler function of the window *hWnd*.

Return values

Return values are application dependent

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpAudio

When to Implement

Objects implementing this interface are real-time audio streams, such as those connected to a live microphone or telephone line. ISpAudio methods allow control over the real-time behavior of the stream. IStream Read and Write methods transfer data to or from an object.

Note: The ISpAudio interface inherits from IStreamFormat.

Methods in Vtable Order

<u>ISpAudio Methods</u>	<u>Description</u>
<u>SetState</u>	Sets the state of the audio device.
<u>SetFormat</u>	Sets the format of the audio device.
<u>GetStatus</u>	Passes back the status of the audio device.
<u>SetBufferInfo</u>	Sets the audio stream buffer

<u>GetBufferInfo</u>	information. Passes back the audio stream buffer information.
<u>GetDefaultFormat</u>	Passes back the default audio format.
<u>EventHandle</u>	Returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.
<u>GetVolumeLevel</u>	Passes back the current volume level.
<u>SetVolumeLevel</u>	Sets the current volume level.
<u>GetBufferNotifySize</u>	Retrieves the audio stream buffer size information.
<u>SetBufferNotifySize</u>	Sets the audio stream buffer size information.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpAudio::EventHandle



- ISpAudio::EventHandle** returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.
- 5 The handle may use one of the various Win32 wait functions, such as `WaitForSingleObject` or `WaitForMultipleObjects`.
- 10 For read streams, set the event when there is data available to read and reset it whenever there is no available data. For write streams, set the event when all of the data has been written to the device, and reset it at any time when there is still data available to be played.
- 15 The caller should not close the returned handle, nor should the caller ever use the event handle after calling `Release()` on the audio object. The audio device will close the handle on the final release of the object.
- 20 **HANDLE EventHandle(void);**

Parameters

None

Return values

Value	Description
HANDLE	Returns valid event handle.

© 1995-2000 Microsoft Corporation. All rights reserved.



ISpVoice

The ISpVoice interface enables an application to perform text synthesis operations.

When to Use

- 5 An application uses the ISpVoice interface to communicate with a SAPI-compliant TTS engine. The ISpVoice::Speak method creates a synthesized output using the engine. It is possible for an application to speak text files or mix synthesized text with
10 audio files in addition to text streams. An application can do this by speaking synchronously or asynchronously.
- Applications can choose a specific TTS voice using the ISpVoice::SetVoice. In order to change the state
15 of the voice (for example, rate, pitch, and volume), use XML inside the ::speak call. Voices can receive different priorities using the ISpVoice::SetPriority. This allows voices with a higher priority to interrupt a voice with a lower priority.
- 20 SAPI returns synthesis events to the application informing the application that the engine has processed a certain event such as bookmarks or phonemes.
- ISpVoice inherits from the ISpEventSource interface.

25 Methods in Vtable Order

ISpVoice Methods

SetOutput

GetOutputObjectToken

GetOutputStream

Pause

Resume

SetVoice

GetVoice

Speak

Description

Sets the current output destination.

Retrieves the current output stream object token.

Retrieves a pointer to an output stream.

Pauses the voice and closes the output device.

Sets the output device to the RUN state and resumes rendering.

Sets the identity of a voice used in text synthesis.

Retrieves the engine voice token information.

Enables the engine to speak the contents of a speak a text buffer.

SpeakStream

Enables the engine to speak the contents of a stream.

GetStatus

Retrieves the current rendering and event status associated with this voice.

Skip

Enables the engine to skip ahead the specified number of items within the current speak request.

SetPriority

Sets the queue priority for a voice.

GetPriority

Retrieves the current voice priority level.

SetAlertBoundary

Specifies which event should be used as the insertion point for alerts.

GetAlertBoundary

Retrieves which event should be used as the insertion point for alerts.

SetRate

Sets the engine's rate of spoken text.

GetRate

Retrieves the engine's rate of spoken text.

SetVolume

Sets the output volume level.

GetVolume

Retrieves the current output volume level.

WaitUntilDone

Specifies the time interval to wait for the speech queue to complete processing.

SetSyncSpeakTimeout

Sets the timeout interval for synchronous speech operations.

GetSyncSpeakTimeout

Retrieves the timeout interval for synchronous speech operations.

SpeakCompleteEvent

Returns an event handle used to wait until the voice has completed speaking.

IsUISupported

Determines if the specified type of UI is supported.

DisplayUI

Displays the requested UI.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpVoice::SpeakCompleteEvent

ISpVoice::SpeakCompleteEvent returns an event handle used to wait until the voice has completed speaking.

- 5 This is similar to the functionality provided by ISpVoice::WaitUntilDone, but allows the caller to

wait on the event handle. The event handle is owned by this object and is not duplicated. The caller must neither call CloseHandle(), nor should the caller ever use the handle after releasing the COM reference to this object.

5 [local] HANDLE SpeakCompleteEvent (void);

Parameters

None.

Return values

	Value	Description
10	Event Handle	For WAIT operation.
	© 1995-2000 Microsoft Corporation. All rights reserved.	

ISpSREngineSite



15 The interface ISpEngineSite is implemented by SAPI and is called by the engine to get audio data and signal detected sound events, and return recognition information.

Methods in Vtable Order

ISpSREngineSite Methods	Description
<u>Read</u>	Reads the input stream in a safe thread method.
<u>DataAvailable</u>	Retrieves the amount of data that can be read.
<u>SetBufferNotifySize</u>	(This method is not yet implemented)
<u>ParseFromTransitions</u>	Parses an ISpPhraseBuilder result from a list of transitions.
<u>Recognition</u>	Indicates an end of the phrase and to start recognition.
<u>AddEvent</u>	Retrieves a RecoContext event handle from the SR engine.
<u>Synchronize</u>	Allows the SR engine to process changes in its active grammar state.
<u>GetWordInfo</u>	Retrieves information for CFG word.
<u>SetWordClientContext</u>	Sets an engine-defined context pointer for a CFG word.
<u>GetRuleInfo</u>	Retrieves information about a CFG rule.
<u>SetRuleClientContext</u>	Sets an engine-defined context pointer for a CFG rule.
<u>GetStateInfo</u>	Retrieves transition state

GetResource

information for CFG transition.
Retrieves a named resource from a grammar.

GetTransitionProperty

Retrieves the SR engine transition property information.

IsAlternate

Determines whether one rule is an alternate of the other.

GetMaxAlternates

Passes back the maximum number of alternates that should be generated for the specified rule.

GetContextMaxAlternates

Passes back the maximum number of alternates that should be generated for the specified recognition context.

UpdateRecoPos

Returns the current position of the recognizer in the stream.

© 1995-2000 Microsoft Corporation. All rights reserved.

ISpSREngineSite::AddEvent

ISpSREngineSite::AddEvent retrieves a RecoContext event handle from the SR engine.

```
5 HRESULT AddEvent(
    const SPEVENT *pEvent,
    SPRECOCONTEXTHANDLE hContext
);
```

Parameters

10 *pEvent*

[in] Address of the SPEVENT structure containing the event information.

hContext

[in] The RecoContext is the event handle passed to SR Engine from SAPI through ISpSREngine::OnCreateRecoContext. A NULL value in *hContext* indicates the event is a global one.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	At least one of <i>pEvent</i> or <i>hContext</i> is invalid or bad. Alternatively, it indicates an event is being added to an inappropriate mode.

E_POINTER	Invalid pointer.
	The current audio stream offset is greater than either the current seek position or the last sync position.
SPERR_STREAM_POS_INVALID	Alternatively, if the event stream is not initialized the stream position is not zero.
FAILED(hr)	Appropriate error message.

© 1995-2000 Microsoft Corporation. All rights reserved.